

# **Отчёт по лабораторной работе №13**

**Дисциплина: Операционные системы**

Мишина Анастасия Алексеевна

# Содержание

<b>1</b>	<b>Цель работы</b>	<b>5</b>
<b>2</b>	<b>Задание</b>	<b>6</b>
<b>3</b>	<b>Теоретическое введение</b>	<b>8</b>
<b>4</b>	<b>Выполнение лабораторной работы</b>	<b>10</b>
<b>5</b>	<b>Вывод</b>	<b>18</b>
<b>6</b>	<b>Ответы на контрольные вопросы</b>	<b>19</b>

# Список иллюстраций

4.1	Создание поддиректории ~/work/os/lab_prog. . . . .	10
4.2	Создание и заполнение файлов. . . . .	10
4.3	Содержимое файла calculate.c. . . . .	11
4.4	Содержимое файла calculate.h. . . . .	11
4.5	Содержимое файла main.c. . . . .	12
4.6	Компиляция программы. . . . .	12
4.7	Содержимое Makefile. . . . .	13
4.8	Запуск отладчика и программы внутри отладчика. . . . .	14
4.9	Просмотр исходного кода постранично и указанных строк. . . . .	14
4.10	Точка останова, просмотр информации и значения переменной . . . . .	15
4.11	Выполнение команды splint calculate.c. . . . .	16
4.12	Выполнение команды splint main.c. . . . .	17

# Список таблиц

# 1 Цель работы

Приобрести простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.

## 2 Задание

1. В домашнем каталоге создайте подкаталог `~/work/os/lab_prog`.
2. Создайте в нём файлы: `calculate.h`, `calculate.c`, `main.c`. Это будет примитивнейший калькулятор, способный складывать, вычитать, умножать и делить, возводить число в степень, брать квадратный корень, вычислять `sin`, `cos`, `tan`. При запуске он будет запрашивать первое число, операцию, второе число. После этого программа выведет результат и остановится.
3. Выполните компиляцию программы посредством `gcc`.
4. При необходимости исправьте синтаксические ошибки.
5. Создайте `Makefile` со заданным содержанием. Поясните в отчёте его содержание.
6. С помощью `gdb` выполните отладку программы `calcul` (перед использованием `gdb` исправьте `Makefile`):
  - Запустите отладчик GDB, загрузив в него программу для отладки:
  - Для запуска программы внутри отладчика введите команду `run`.
  - Для постраничного (по 9 строк) просмотра исходного код используйте команду `list`.
  - Для просмотра строк с 12 по 15 основного файла используйте `list` с параметрами.

- Для просмотра определённых строк не основного файла используйте `list` с параметрами.
  - Установите точку останова в файле `calculate.c` на строке номер 21.
  - Выведите информацию об имеющихся в проекте точка останова.
  - Запустите программу внутри отладчика и убедитесь, что программа остановится в момент прохождения точки останова.
  - Посмотрите, чему равно на этом этапе значение переменной `Numeral`, введя `print Numeral`.
  - Сравните с результатом вывода на экран после использования команды: `display Numeral`.
  - Уберите точки останова.
- 7. С помощью утилиты `splint` попробуйте проанализировать коды файлов `calculate.c` и `main.c`.

## 3 Теоретическое введение

Процесс разработки программного обеспечения обычно разделяется на следующие этапы:

- планирование, включающее сбор и анализ требований к функционалу и другим характеристикам разрабатываемого приложения;
- проектирование, включающее в себя разработку базовых алгоритмов и спецификаций, определение языка программирования;
- непосредственная разработка приложения:
- кодирование — по сути создание исходного текста программы (возможно в нескольких вариантах);
- анализ разработанного кода;
- сборка, компиляция и разработка исполняемого модуля;
- тестирование и отладка, сохранение произведённых изменений;
- документирование.

Для создания исходного текста программы разработчик может воспользоваться любым удобным для него редактором текста: vi, vim, mceditor, emacs, geany и др.

После завершения написания исходного кода программы (возможно состоящей из нескольких файлов), необходимо её скомпилировать и получить исполняемый модуль.

Отладка — этап разработки компьютерной программы, на котором обнаруживают, локализуют и устраняют ошибки. Чтобы понять, где возникла



ошибка, приходится:

- узнавать текущие значения переменных;
- выяснять, по какому пути выполнялась программа.

## 4 Выполнение лабораторной работы

1. Создаю поддиректорию `~/work/os/lab_prog` (рис. [4.1]).:

```
[aamishina@fedora ~]$ cd ~/work/os  
[aamishina@fedora os]$ mkdir lab_prog  
[aamishina@fedora os]$ cd lab_prog/  
[aamishina@fedora lab_prog]$ pwd  
/home/aamishina/work/os/lab_prog
```

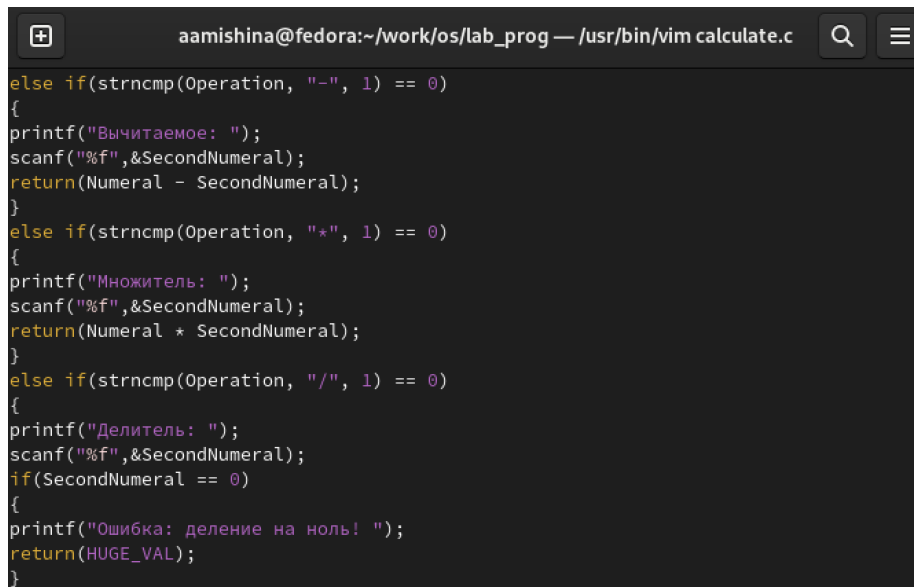
Рис. 4.1: Создание поддиректории `~/work/os/lab_prog`.

2. Создаю файлы `calculate.h`, `calculate.c`, `main.c` и заполняю их, согласно описанию лабораторной работы (рис. [4.2]).:

```
[aamishina@fedora lab_prog]$ touch calculate.c calculate.h main.c  
[aamishina@fedora lab_prog]$ ls  
calculate.c calculate.h main.c
```

Рис. 4.2: Создание и заполнение файлов.

Это будет примитивнейший калькулятор, способный складывать, вычитать, умножать и делить, возводить число в степень, брать квадратный корень, вычислять `sin`, `cos`, `tan`. При запуске он будет запрашивать первое число, операцию, второе число. После этого программа выведет результат и остановится (рис. [4.3]), (рис. [4.4]), (рис. [4.5]).



```
else if(strncmp(Operation, "-", 1) == 0)
{
printf("Вычитаемое: ");
scanf("%f",&SecondNumeral);
return(Numeral - SecondNumeral);
}
else if(strncmp(Operation, "*", 1) == 0)
{
printf("Множитель: ");
scanf("%f",&SecondNumeral);
return(Numeral * SecondNumeral);
}
else if(strncmp(Operation, "/", 1) == 0)
{
printf("Делитель: ");
scanf("%f",&SecondNumeral);
if(SecondNumeral == 0)
{
printf("Ошибка: деление на ноль! ");
return(HUGE_VAL);
}
```

Рис. 4.3: Содержимое файла calculate.c.



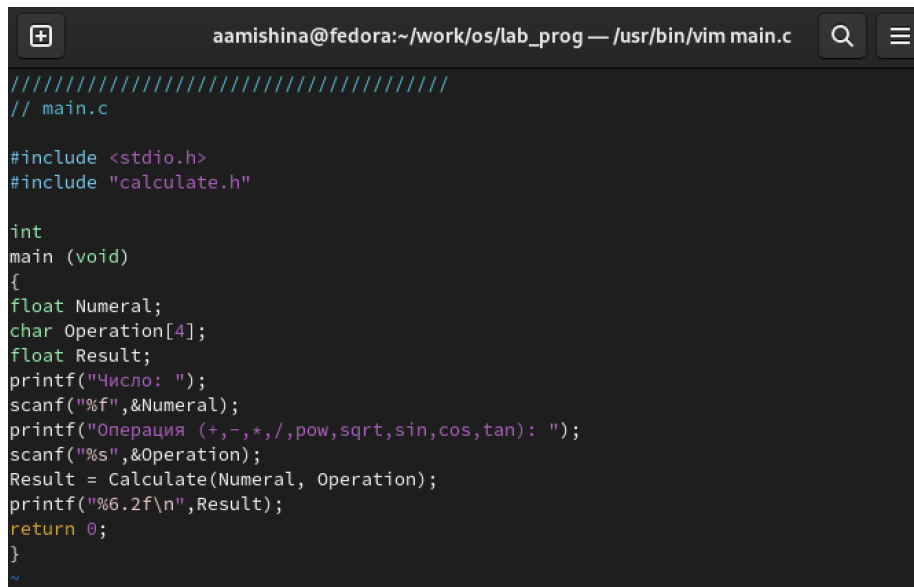
```
////////////////////////////////////
// calculate.h

#ifndef CALCULATE_H_
#define CALCULATE_H_

float Calculate(float Numeral, char Operation[4]);

#endif /*CALCULATE_H_*/
~
```

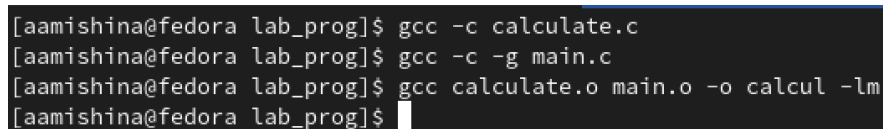
Рис. 4.4: Содержимое файла calculate.h.



```
#####  
// main.c  
  
#include <stdio.h>  
#include "calculate.h"  
  
int  
main (void)  
{  
    float Numeral;  
    char Operation[4];  
    float Result;  
    printf("Число: ");  
    scanf("%f",&Numeral);  
    printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");  
    scanf("%s",&Operation);  
    Result = Calculate(Numeral, Operation);  
    printf("%6.2f\n",Result);  
    return 0;  
}
```

Рис. 4.5: Содержимое файла main.c.

3. Выполняю компиляцию программы посредством gcc (рис. [4.6]):



```
[aamishina@fedora lab_prog]$ gcc -c calculate.c  
[aamishina@fedora lab_prog]$ gcc -c -g main.c  
[aamishina@fedora lab_prog]$ gcc calculate.o main.o -o calcul -lm  
[aamishina@fedora lab_prog]$
```

Рис. 4.6: Компиляция программы.

4. Создаю. Makefile с содержанием, согласно описанию лабораторной работы, при этом немного изменяю его (рис. [4.7]).:

```
#
# Makefile
#

CC=gcc
CFLAGS=-g
LIBS=-lm

calcul: calculate.o main.o
gcc calculate.o main.o -o calcul $(LIBS)

calculate.o: calculate.c calculate.h
gcc -c calculate.c $(CFLAGS)

main.o: main.c calculate.h
gcc -c main.c $(CFLAGS)

clean:
-rm calcul *.o

# End Makefile
~
```

Рис. 4.7: Содержимое Makefile.

В Makefile указываю компилятор gcc, флаг -g и дополнительные библиотеки -lm. Описываю, какие команды необходимо запустить, чтобы получить файлы calcul, calculate.o и main.o, подключив дополнительные библиотеки и флаги. А в поле clean описывается удаление файлов calcul и файлов, оканчивающихся на “.o”.

5. С помощью gdb выполняю отладку программы calcul:

Запускаю отладчик GDB, загрузив в него программу для отладки.

Запускаю программу внутри отладчика, введя команду run (рис. [4.8]).:

```
[aamishina@fedora lab_prog]$ gdb ./calcul
GNU gdb (GDB) Fedora Linux 12.1-7.fc37
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "aarch64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./calcul...

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.fedoraproject.org/
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
(No debugging symbols found in ./calcul)
(gdb) run
Starting program: /home/aamishina/work/os/lab_prog/calcul
Downloading 0.03 MB separate debug info for system-supplied DSO at 0xffffffff7ffb000
Downloading 1.63 MB separate debug info for /lib64/libm.so.6
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): pow
Степень: 2
    25.00
[Inferior 1 (process 4678) exited normally]
```

Рис. 4.8: Запуск отладчика и программы внутри отладчика.

Постранично просматриваю исходный код с помощью команды `list`.

Просматриваю строки с 12 по 15 основного файла с помощью команды `list` с параметрами (рис. [4.9]):

```
(gdb) list
1  //////////////////////////////////////////////////
2  // main.c
3
4  #include <stdio.h>
5  #include "calculate.h"
6
7  int
8  main (void)
9  {
10     float Numeral;
(gdb) list 12,15
12     float Result;
13     printf("Число: ");
14     scanf("%f",&Numeral);
15     printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
(gdb) █
```

Рис. 4.9: Просмотр исходного кода постранично и указанных строк.

Определённые строки не основного файла с помощью команды `list` с параметрами посмотреть не удалось.

Устанавливаю точку останова в файле `main.c` на строке номер 17 и вывожу информацию об имеющихся в проекте точках останова. Запускаю программу внутри отладчика и убеждаюсь, что программа остановилась в момент прохождения точки останова, затем с помощью команды `backtrace` просматриваю весь стек вызываемых функций от начала программы до текущего места. Просматриваю, чему равно значение переменной `Numeral`, введя сначала `print Numeral`, она равна 5, а затем сравниваю с выводом команды `display Numeral`. Можем заметить, что выходы разные, но значение одно - 5. Затем удаляю точку останова (рис. [4.10]):

```
(gdb) break 17
Breakpoint 1 at 0x400bd0: file main.c, line 17.
(gdb) info breakpoints
Num     Type             Disp Enb Address            What
1       breakpoint      keep y   0x0000000000400bd0 in main at main.c:17
(gdb) run
Starting program: /home/aamishina/work/os/lab_prog/calcul

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.fedoraproject.org/
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -

Breakpoint 1, main () at main.c:17
17      Result = Calculate(Numeral, Operation);
(gdb) backtrace
#0  main () at main.c:17
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
(gdb) info breakpoints
Num     Type             Disp Enb Address            What
1       breakpoint      keep y   0x0000000000400bd0 in main at main.c:17
        breakpoint already hit 1 time
(gdb) delete 1
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb)
```

Рис. 4.10: Точка останова, просмотр информации и значения переменной

6. С помощью заранее установленной утилиты `splint` анализирую коды

файла calculate.c (вижу 15 предупреждений) и файла main.c (вижу 4 предупреждения) (рис. [4.11]), (рис. [4.12]):

```
[aamishina@fedora lab_prog]$ splint calculate.c
Splint 3.1.2 --- 23 Jul 2022

calculate.h:7:37: Function parameter Operation declared as manifest array (size
                    constant is meaningless)
    A formal parameter is declared as an array with size. The size of the array
    is ignored in this context, since the array formal parameter is treated as a
    pointer. (Use -fixedformalarray to inhibit warning)
calculate.c:10:31: Function parameter Operation declared as manifest array
                    (size constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:16:1: Return value (type int) ignored: scanf("%f", &Sec...
    Result returned by function call is not used. If this is intended, can cast
    result to (void) to eliminate message. (Use -retvalint to inhibit warning)
calculate.c:22:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:28:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:35:4: Dangerous equality comparison involving float types:
                    SecondNumeral == 0
    Two real (float, double, or long double) values are compared directly using
    == or != primitive. This may produce unexpected results since floating point
    representations are inexact. Instead, compare the difference to FLT_EPSILON
    or DBL_EPSILON. (Use -realcompare to inhibit warning)
calculate.c:38:7: Return value type double does not match declared type float:
                    (HUGE_VAL)
    To allow all numeric types to match, use +relaxtypes.
calculate.c:46:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:47:7: Return value type double does not match declared type float:
                    (pow(Numeral, SecondNumeral))
calculate.c:50:7: Return value type double does not match declared type float:
                    (sqrt(Numeral))
calculate.c:52:7: Return value type double does not match declared type float:
                    (sin(Numeral))
calculate.c:54:7: Return value type double does not match declared type float:
                    (cos(Numeral))
calculate.c:56:7: Return value type double does not match declared type float:
                    (tan(Numeral))
calculate.c:60:7: Return value type double does not match declared type float:
                    (HUGE_VAL)

Finished checking --- 15 code warnings
[aamishina@fedora lab_prog]$
```

Рис. 4.11: Выполнение команды splint calculate.c.



```

[aamishina@fedora lab_prog]$ splint main.c
Splint 3.1.2 --- 23 Jul 2022

calculate.h:7:37: Function parameter Operation declared as manifest array (size
                    constant is meaningless)
    A formal parameter is declared as an array with size. The size of the array
    is ignored in this context, since the array formal parameter is treated as a
    pointer. (Use -fixedformalarray to inhibit warning)
main.c: (in function main)
main.c:14:1: Return value (type int) ignored: scanf("%f", &Num...
    Result returned by function call is not used. If this is intended, can cast
    result to (void) to eliminate message. (Use -retvalint to inhibit warning)
main.c:16:12: Format argument 1 to scanf (%s) expects char * gets char [4] *:
    &Operation
    Type of parameter is not consistent with corresponding code in format string.
    (Use -formattype to inhibit warning)
    main.c:16:9: Corresponding format code
main.c:16:1: Return value (type int) ignored: scanf("%s", &Ope...

Finished checking --- 4 code warnings
[aamishina@fedora lab_prog]$

```

Рис. 4.12: Выполнение команды splint main.c.

## 5 Вывод

В ходе выполнения лабораторной работы я приобрела простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.

## 6 Ответы на контрольные вопросы

1. Как получить более полную информацию о программах: gcc, make, gdb и др.?

Дополнительную информацию о этих программах можно получить с помощью функций `info` и `man`.

2. Назовите и дайте краткую характеристику основным этапам разработки приложений в UNIX?

Unix поддерживает следующие основные этапы разработки приложений:

- создание исходного кода программы;

- представляется в виде файла;

- сохранение различных вариантов исходного текста;

- анализ исходного текста; Необходимо отслеживать изменения исходного кода, а также при работе более двух программистов над проектом программы нужно, чтобы они не делали изменений кода в одно время.

- компиляция исходного текста и построение исполняемого модуля;

- тестирование и отладка;

- проверка кода на наличие ошибок

- сохранение всех изменений, выполняемых при тестировании и отладке.

3. Что такое суффиксы и префиксы? Основное их назначение. Приведите примеры их использования.

Суффиксы и префиксы указывают тип объекта. Использование суффик-

са “.с” для имени файла с программой на языке Си отражает удобное и полезное соглашение, принятое в ОС UNIX. Для любого имени входного файла суффикс определяет какая компиляция требуется.

Одно из полезных свойств компилятора Си — его способность по суффиксам определять типы файлов.

По суффиксу .с компилятор распознает, что файл abcd.c должен компилироваться, а по суффиксу .о, что файл abcd.o является объектным модулем и для получения исполняемой программы необходимо выполнить редактирование связей. Простейший пример командной строки для компиляции программы abcd.c и построения исполняемого модуля abcd имеет вид: gcc -o abcd abcd.c.

Некоторые проекты предпочитают показывать префиксы в начале текста изменений для старых (old) и новых (new) файлов. Опция – prefix может быть использована для установки такого префикса. Плюс к этому команда bzr diff -p1 выводит префиксы в форме которая подходит для команды patch -p1.

#### 4. Основное назначение компилятора с языка Си в UNIX?

Основное назначение компилятора с языка Си заключается в компиляции всей программы в целом и получении исполняемого модуля.

#### 5. Для чего предназначена утилита make.

При разработке большой программы, состоящей из нескольких исходных файлов заголовков, приходится постоянно следить за файлами, которые требуют перекомпиляции после внесения изменений. Программа make освобождает пользователя от такой рутинной работы и служит для документирования взаимосвязей между файлами. Описание взаимосвязей и соответствующих действий хранится в так называемом make-файле, который по умолчанию имеет имя makefile или Makefile.

#### 6. Приведите структуру make-файла. Дайте характеристику основным

элементам этого файла.

makefile может иметь следующий вид:

```
#  
#  
Makefile  
#  
CC = gcc  
CFLAGS =  
LIBS = -lm  
calcul: calculate.o main.o  
gcc calculate.o main.o -o calcul $(LIBS)  
calculate.o: calculate.c calculate.h  
gcc -c calculate.c $(CFLAGS)  
main.o: main.c calculate.h  
gcc -c main.c $(CFLAGS)  
clean: -rm calcul .o ~  
# End Makefile
```

В общем случае make-файл содержит последовательность записей (строк), определяющих зависимости между файлами.

Первая строка записи представляет собой список целевых (зависимых) файлов, разделенных пробелами, за которыми следует двоеточие и список файлов, от которых зависят целевые. Текст, следующий за точкой с запятой, и все последующие строки, начинающиеся с литеры табуляции, являются командами ОС UNIX, которые необходимо выполнить для обновления целевого файла. Таким образом, спецификация взаимосвязей имеет формат: target1 [ target2...]: [:] [dependment1...] [(tab)commands] [#commentary] [(tab)commands] [#commentary], где # — специфицирует начало комментария, так как содержимое строки, начиная с # и до конца строки, не будет

обрабатываться командой `make`; `:` — последовательность команд ОС UNIX должна содержаться в одной строке `make`-файла (файла описаний), есть возможность переноса команд `()`, но она считается как одна строка; `::` — последовательность команд ОС UNIX может содержаться в нескольких последовательных строках файла описаний.

Приведённый выше `make`-файл включает два способа компиляции и построения исполняемого модуля.

Первый способ предусматривает обычную компиляцию с построением исполняемого модуля с именем `abcd`.

Второй способ позволяет включать в исполняемый модуль `testabcd` возможность выполнить процесс отладки на уровне исходного текста.

7. Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать?

Пошаговая отладка программ заключается в том, что выполняется один оператор программы и, затем контролируются те переменные, на которые должен был воздействовать данный оператор. Если в программе имеются уже отлаженные подпрограммы, то подпрограмму можно рассматривать, как один оператор программы и воспользоваться вторым способом отладки программ. Если в программе существует достаточно большой участок программы, уже отлаженный ранее, то его можно выполнить, не контролируя переменные, на которые он воздействует. Использование точек останова позволяет пропускать уже отлаженную часть программы. Точка останова устанавливается в местах, где необходимо проверить содержимое переменных или просто проконтролировать, передаётся ли управление данному оператору. Практически во всех отладчиках поддерживается это свойство (а также выполнение программы до курсора и выход из подпрограммы). Затем отладка программы продолжается в пошаговом режиме с контролем локальных и глобальных переменных, а

также внутренних регистров микроконтроллера и напряжений на выводах этой микросхемы.

#### 8. Назовите и дайте основную характеристику основным командам отладчика gdb.

`backtrace` – выводит весь путь к текущей точке останова, то есть названия всех функций, начиная от `main()`; иными словами, выводит весь стек функций;

`break` – устанавливает точку останова; параметром может быть номер строки или название функции;

`clear` – удаляет все точки останова на текущем уровне стека (то есть в текущей функции);

`continue` – продолжает выполнение программы от текущей точки до конца;

`delete` – удаляет точку останова или контрольное выражение;

`display` – добавляет выражение в список выражений, значения которых отображаются каждый раз при остановке программы;

`finish` – выполняет программу до выхода из текущей функции; отображает возвращаемое значение, если такое имеется;

`info breakpoints` – выводит список всех имеющихся точек останова; – `info watchpoints` – выводит список всех имеющихся контрольных выражений;

`splist` – выводит исходный код; в качестве параметра передаются название файла исходного кода, затем, через двоеточие, номер начальной и конечной строки; `next` – пошаговое выполнение программы, но, в отличие от команды `step`, не выполняет пошагово вызываемые функции;

`print` – выводит значение какого-либо выражения (выражение передается в качестве параметра);

`run` – запускает программу на выполнение;

set – устанавливает новое значение переменной

step – пошаговое выполнение программы;

watch – устанавливает контрольное выражение, программа остановится, как только значение контрольного выражения изменится;

9.Опишите по шагам схему отладки программы которую вы использовали при выполнении лабораторной работы.

1. Я скомпилировала программу

2. Выполнила ее

3. Затем я воспользовалась командами list, list 12,15, list calculate.c:20,29 и другими

4. Поставила точку останова на некоторой строке

5. Вновь запустила программу

6. Удалила точку останова.

10.Прокомментируйте реакцию компилятора на синтаксические ошибки в программе при его первом запуске.

У меня не было никаких синтаксических ошибок при первом запуске.

11.Назовите основные средства, повышающие понимание исходного кода программы.

Если вы работаете с исходным кодом, который не вами разрабатывался, то назначение различных конструкций может быть не совсем понятным. Система разработки приложений UNIX предоставляет различные средства, повышающие понимание исходного кода. К ним относятся: – cscope - исследование функций, содержащихся в программе; – splint — критическая проверка программ, написанных на языке Си.

12.Каковы основные задачи, решаемые программой slint?



Проверка корректности задания аргументов всех использованных в программе функций, а также типов возвращаемых ими значений;

Поиск фрагментов исходного текста, корректных с точки зрения синтаксиса языка Си, но малоэффективных с точки зрения их реализации или содержащих в себе семантические ошибки;

Общая оценка мобильности пользовательской программы.